Proceedings of the 2023 Winter Simulation Conference
C. G. Corlu, S. R. Hunter, H. Lam, B. S. Onggo, J. Shortle, and B. Biller, eds.

NBSIMGEN: JUPYTER NOTEBOOK EXTENSION FOR GENERATING SIMULATION EXPERIMENTS

Pia Wilsdorf
Anton Willy Kirchhübel
Adelinde M. Uhrmacher

Institute for Visual and Analytic Computing
University of Rostock
Albert-Einstein-Str. 22
Rostock, 18059, GERMANY

ABSTRACT

Simulation experiments are crucial in conducting simulation studies. With simulation studies growing increasingly complex, simulation experiments are intertwined with steps of conceptual modeling, model building, analyzing data, and visualizing and interpreting results. Making the products of these various steps (assumptions, requirements, data, model components, and experiments) explicit has been shown to increase the reproducibility of simulation studies. Moreover, using an integrated environment that allows developing, organizing and documenting those products can facilitate their automatic reuse and exploitation. We explore Jupyter Notebook as an all-in-one solution for conducting and documenting a simulation study, and we present nbSimGen. This Jupyter Notebook extension lends support to modelers by automatically specifying and running suitable simulation experiments. It is based on an annotation vocabulary that, during the development of the conceptual model and the simulation model, allows users to mark portions of their notebook deemed relevant to the various simulation experiments to come.

1 INTRODUCTION

Due to the ever-expanding technological progress in our society, the demand for simulation studies to prove hypotheses and to gain further insights is steadily growing. Simulation studies can be applied in various fields. For example, they can support traffic planning by analyzing the traffic flow of an intersection, or they can lend decision support in political crises, such as the recent COVID-19 pandemic.

Simulation studies in those areas usually turn out to be very complex since many factors have to be considered for the outcome of such an event. For example, the analysis of a traffic intersection requires, among other things, exact data on the number of road users at all times of the day or year and their driving behavior, and exact information about the traffic lights and the overall road network, so that relationships between intersections may also be taken into account. Thus, numerous steps of conceptual modeling, building the simulation model, analyzing data, specifying and executing simulation experiments, and visualizing and interpreting results are required and closely intertwined.

To minimize the risk that wrong conclusions are drawn for real systems, which can have drastic consequences depending on the subject of modeling, and to avoid that study results cannot be reproduced by other researchers in the field, the demand for comprehensive documentation of simulation studies is also increasing. Documentations should ideally cover the entire development process of the simulation models, including their relation to simulation experiments and the corresponding data and model assumptions and requirements, as underpinned by the various reporting guidelines (Grimm et al. 2014) and provenance standards (Ruscheinski and Uhrmacher 2017).

979-8-3503-6966-3/23/$31.00 ©2023 IEEE

The solution taken, e.g., by the COMBINE archive (Bergmann et al. 2014) or the Workflow Research Objects (Carvalho et al. 2016) is to store all the artifacts of a simulation study together in one place, including all the models, scripts, data, research questions, assumptions, and requirements. In addition, modeling notebooks can provide a logbook of the activities conducted and the rationale behind them (Ayllón et al. 2021), which would further increase the reproducibility of the simulation studies and make them easier to understand. Furthermore, an integrated, easy-to-use environment would facilitate the reuse and automatic exploitation of existing information and artifacts collected and created during the simulation study. For example, the conduction of simulation experiments, which form a salient feature of the modeling and simulation life cycle (Balci 2012; Sargent 2013), may be partly automated, saving modelers time and effort during the calibration, validation, and analysis of simulation models as well as during prediction.

This paper explores Jupyter Notebook as an all-in-one solution to conduct and store everything about a simulation study. Due to the straightforward, web-based design and the largely language-independent usability, Jupyter Notebook already enjoys great popularity for conducting scientific studies (Beg et al. 2021). We present *nbSimGen*, a Jupyter Notebook extension that further supports modelers by automatically specifying and running suitable simulation experiments. The automatic experiment generation in *nbSimGen* works based on an annotation vocabulary that, during the development of the conceptual model and the simulation model, allows users to mark portions of their notebook deemed relevant to the various simulation experiments to come. This allows the extension to distinguish the different ingredients of a simulation experiment from each other, and to interpret and integrate them correctly into an executable simulation experiment, automatically selecting suitable methods and libraries for the task at hand. In developing *nbSimGen*, we make the following contributions:

1. A guide for conducting and documenting simulation studies inside a modeling notebook.
2. A vocabulary for annotating information relevant to the simulation experiments either in the code or accompanying text.
3. An experiment generator that evaluates the annotated information and generates executable code for various types of simulation experiments.
4. A built-in decision support for selecting the methods and libraries to use.
5. A case study in which the approach is demonstrated based on a microscopic traffic simulation.

The remainder of the paper is organized as follows. In Section 2, we provide an overview of Jupyter Notebook and code generation applied in simulation studies. Section 3 is dedicated to the overall architecture of the Jupyter Notebook extension *nbSimGen*. In Section 4, we present the approach in a microscopic traffic simulation case study. This is followed by conclusions and future work in Section 5.

2 BACKGROUND

2.1 Jupyter Notebook and its Extensions

Jupyter Notebook is a popular browser-based open-source tool that provides a virtual interactive notebook, which offers the user the possibility to write code, documentation, data and visualizations (Kluyver et al. 2016). These notebooks often reside in open online repositories and interface other objects such as data sets, code or publications stored in other locations (Randles et al. 2017). A Jupyter Notebook consists of a collection of different kinds of cells through which input is provided to the notebook. Code cells can be used to write new or edit existing code. The code is interpreted according to the selected kernel and the output is rendered inside the notebook. The default kernel interprets Python code, however, different programming languages may be used. Markdown cells can be used to format the notebooks and provide additional information, such as an outline of the research question as verbal narrative, a sketch of the processes to be included in the model, or a list of assumptions. Lastly, raw cells are only rendered when exporting the notebook to another format, such as HTML or LaTeX, using the nbConvert tool. Raw cells are rarely used and will therefore not be considered in this paper.

Jupyter Notebook comes with a minimalist set of functionalities. Jupyter Notebook Extensions thus allow extending the regular Jupyter Notebook environment. An extension may improve ease-of-use and productivity of users, e.g., by automatically generating a table of contents for the written cells or by autoformatting Python code. Furthermore, they can enhance the understanding of the code as well as the quality and reproducibility of the Jupyter notebooks (Pimentel et al. 2021).

2.2 Jupyter Notebook in Simulation Studies

Jupyter Notebook has been shown to confer several advantages for the conduction of simulation studies. First, they possess the capabilities for processing large data sets. Simulation output data can be generated and processed directly within a notebook, thus the notebooks are particularly suited for data-driven modeling and simulation (Savira et al. 2021). Second, notebooks can be easily exported into different file formats, which improves the readability of the entire scientific study. In addition, studies conducted with Jupyter notebooks have all relevant information, including code, comments and links to other sources, within one file, improving the reproducibility and credibility of modeling and simulation studies (Beg et al. 2021). Analogously to laboratory notebooks, Juypter notebooks, as a form of electronic modeling notebooks, allow users to create a log of their daily activities and all the associated data and code (Ayllón et al. 2021). Furthermore, Jupyter notebooks provide an integrated environment for modeling and data analysis, which can be tailored to a specific use case by installing the necessary Python packages or Docker images. The CoLoMoTo interactive notebook is an example of such pre-programmed content, as it provides an environment for editing qualitative models of biological networks (Naldi et al. 2018). Similarly, Tellurium provides a frontend for quantitative modeling in the field of systems biology using the various standardized formats of that community (Medley et al. 2018). Simulation platforms, such as SIMEX (Fangohr et al. 2019), also use Jupyter as a frontend.

To promote the use of Juypter for large and complex simulation studies, further simulation-specific assistance will be required that takes information from the whole notebook into account to automate tasks, such as the automatic generation of simulation experiments.

2.3 Generation of Simulation Experiments

The design and execution of simulation experiments has been identified as an essential phase of the life cycle of simulation studies. Depending on the type of simulation experiment (e.g., parameter scanning, sensitivity analysis, simulation-based optimization or statistical model checking), initial parameter configurations or parameter bounds as well as configurations of the experiment design or analysis have to be set. By running the experiment, the input configurations are transformed into outputs, which should provide insights about the real system. Experiment generation aims to set the inputs of an experiment (the model parameters as well as the observables and other experiment-specific configurations) automatically and to produce code accordingly. To accomplish this, it may take additional information about the simulation study into account. Formally specified hypotheses may, e.g., be used to generate an experiment design that covers the parameter space necessary to perform automatic hypothesis testing (Lorig 2019). In addition to temporal-logic formulae, data provided in a semi-structured format (such as tables) can be used to derive input configurations (Ruscheinski et al. 2018). To be able to generate experiment code for a variety of specification languages, model-driven approaches have been explored. Those allow generating code in various domain-specific and general-purpose languages from a generalized intermediary representation (also known as metamodel), and thus flexibly designing multiple types of simulation experiments (Wilsdorf et al. 2022).

So far, automatic experiment generation has been explored and demonstrated in several case studies, but it is not yet available as a built-in feature of popular modeling and analysis environments, such as Jupyter Notebook, and therefore not easily integrated in the typical workflow of a modeler, or data scientist more generally.

2.4 Code Generation in Jupyter Notebooks

Code generation has also been applied in Jupyter notebooks in the context of data science. For example, a tool exists that automatically transforms and cleans up code (Head et al. 2019). A code snippet library can also assist in efficiently creating clean code (Mooers 2021). Template-based approaches are a popular choice for code generation, e.g., in generating machine learning pipelines from user-specified metadata (Mustafa et al. 2023). In a programming-by-example approach, input-output pairs are provided to a notebook cell based on which code for typical data science tasks is synthesized (Drosos et al. 2020). Code can also be synthesized from a natural language description of the data scientist's goal (Yin et al. 2022). Furthermore, by exploring knowledge of user interaction patterns, specific cells can be migrated from local execution in the notebook to a more suitable high-performance cloud platform (Cunha et al. 2021). In addition, annotations of Jupyter notebooks have been proposed for enabling understanding and reproducibility of the notebooks (Rule et al. 2019). An approach that exploits a variety of information contained in a notebook extensively and focuses on the needs of modelers and simulationists, however, has been elusive.

3 THE NBSIMGEN JUPYTER NOTEBOOK EXTENSION

The modeling notebook, given by a Jupyter notebook, is the central element of our approach, as it presents a "history" of the simulation study in terms of the data collected, the model implemented, and the analyses conducted. From this history—together with semantic annotations—the developed Jupyter Notebook extension, named *nbSimGen*, automatically builds adequate simulation experiments.

3.1 Envisioned Workflow

The workflow we envision for modelers is presented in Figure 1 (left-hand side). At the beginning of a new simulation study, the modeler creates a new Juypter notebook. Typically, a simulation study starts by formulating the research question, assumptions, requirements as well as collecting several materials such as papers, data, or previous models. Based on that information, a first, conceptualized draft of the model is created including what components or parameters the model will have. This may be accompanied by descriptions in natural language as well as first code snippets defining variables or data structures, and thus consists of a collection of Markdown and code cells.

What we just described is also known as the conceptual modeling phase (Robinson 2017). Here, annotations of *nbSimGen* can be used to mark, e.g., an important parameter for further analysis or to indicate which parameter values to exclude. After the conceptual model has been assembled, the actual simulation model is implemented. This may comprise the development of a simulator if no ready-made simulation library, such as SimPy (Matloff 2008), is used that only requires setting up the model and simulator configurations. Similarly to the conceptual model, the model code should be documented and annotated with certain keywords.

At some point during model (and simulator) building, the modeler will want to run analyses to check if the behavior of the model meets the modeling requirements (e.g., validation using statistical model checking), explore the parameter space (analysis using a parameter scan), find parameter values for which the model yields satisfying results w.r.t. some expectation (e.g., calibration using approximate Bayesian methods), or to find optimal parameter values w.r.t. the research question (simulation-based optimization). To run any of these mentioned simulation experiments, the modeler can invoke the experiment generator *nbSimGen* via the graphical user interface embedded inside Jupyter. The *nbSimGen* will retrieve the annotated data, and if enough information is already contained in the notebook, code of a simulation experiment will be generated in a new code cell. The details of that entire generation process will be described in the following.

Before running the experiment, the user may adapt the code. This step is necessary to keep the modeler in control of the experiments, and thus the simulation study. The experiment is executable inside the notebook, and thus results, plots, and additional evaluations can be directly conducted in the same environment given by Jupyter. Based on the results, the modeler can refine the conceptual model as well as

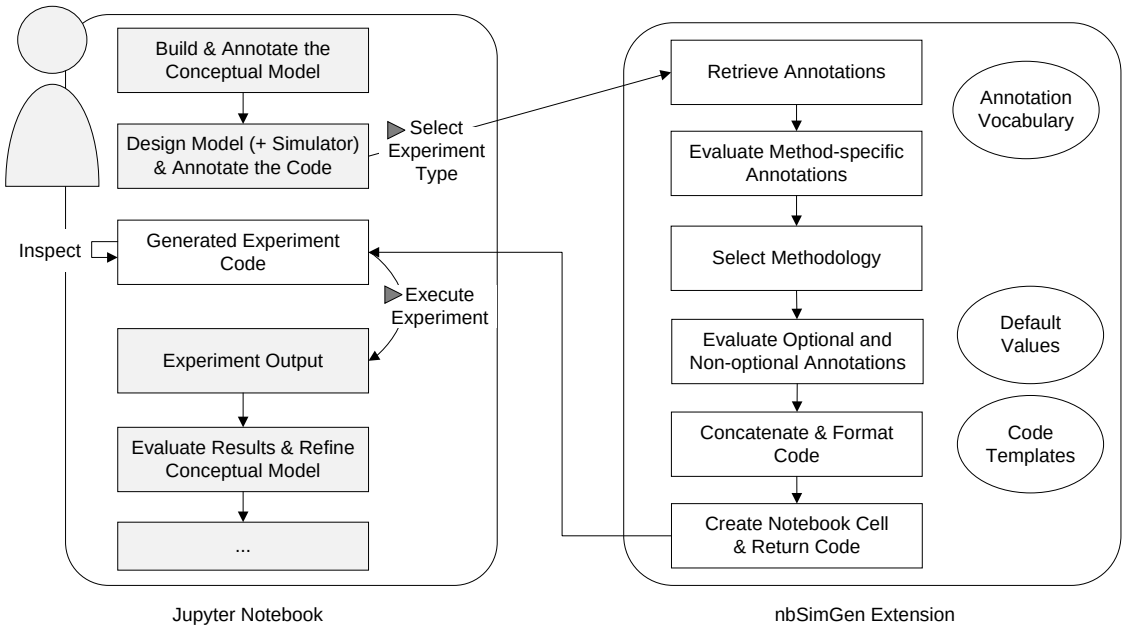

Figure 1: Workflow of a user when conducting a simulation study in Jupyter Notebook (left, grey) and architecture of the *nbSimGen* extension (right). Selection of an experiment type by the user triggers the automatic experiment generation of the notebook extension, which adds a new cell to the notebook (left, white) containing code for the requested simulation experiment.

the simulation model, e.g., by collecting additional data or by revising research questions and constraints. This starts a new modeling and simulation cycle that again can be supported via *nbSimGen*.

3.2 Guidelines and Assumptions

We aim to support a variety of experiment types, building on the methods from various libraries. Currently, *nbSimGen* focuses on support for Jupyter notebooks using the Python kernel and thus only Python libraries are used as part of the code generation. Aside from the currently supported parameter scans and simulation-based optimization algorithms that rely on the SciPy library, the notebook extension is designed to be expanded flexibly with other experiment types (e.g., statistical model checking or sensitivity analysis) and to coordinate access to useful Python packages.

The users of *nbSimGen* are expected to have basic knowledge about Python and Jupyter Notebook—ideally they have conducted simulation studies in this environment before. Furthermore, they need to follow simple guidelines for annotating and sectioning their notebook.

The notebook should be kept as the user proceeds and thus intertwine code, information collection, and documentations. Ideally, for each step (e.g., introducing a new data source in the conceptual model, a new model component or a new analysis), a new cell should be created at the bottom of the notebook to maintain the order in which things have been developed and for cells to be executed in the correct order when running the entire notebook. Further, we require the user to annotate their notebook using a curated vocabulary as they go.

In addition to these annotations, for the experiment generator to be able to identify the simulator and to start simulation runs, we expect a main simulation function named `run_simulation` to be implemented in one of the code cells that takes a dictionary containing the model parameter names and values and the desired output variables as arguments, then runs the model with that particular parameter configuration, and finally returns a list of outputs. In the case of stochastic simulation, the returned result should represent a summary statistic of the several stochastic replications. Due to the use of the wrapper function, the

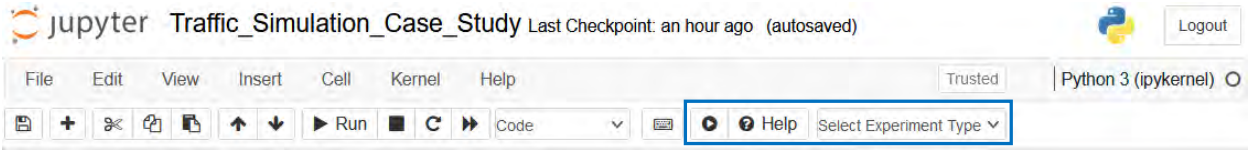

Figure 2: Extended Jupyter toolbar (blue frame). The experiment type is selected via the drop-down options, and the play button allows generating the simulation experiment. The Help button opens the tool's documentation.

experiment generation with *nbSimGen* caters equally to users of ready-made simulators, such as SimPy for discrete-event simulation (Matloff 2008) or Tellurium for systems biology applications (Medley et al. 2018), and users of custom simulators (as used in our case study).

3.3 Graphical User Interface

The interface of Jupyter Notebook is extended by three control elements: a drop-down selection and two buttons, which are shown in Figure 2. The selector allows the user to choose the type of the next simulation experiment to be conducted next in the study. To generate the selected experiment in a code cell, a new play button exists. To run the generated code, the familiar Jupyter functionality "Run and Select Below" can be used, which will run the code and subsequently prepare a new cell below. The help menu provides assistance for keeping the modeling notebook and applying the annotation vocabulary.

3.4 Annotation Vocabulary and Decision Making

Within *nbSimGen*, we use annotations to mark crucial information inside the notebook. The annotation terms always start with an "@" and can be placed both in code cells and Markdown cells. In code cells, they have to be included as part of a line comment, i.e., behind the symbol "#".

In previous work, we have already developed templates and metamodels that can capture the key features of a simulation experiment in a generalized vocabulary (Ruscheinski et al. 2018; Wilsdorf et al. 2022). This also allowed us to find commonalities and differences between experiment types (i.e., the required and optional inputs). This vocabulary was now refined to more specifically support different kinds of parameter scans and optimization algorithms. For parameter scans, we currently use the vocabulary listed in Table 1. The vocabulary for simulation-based optimization can be viewed in our GitHub repository. Note, however, that we do not consider these as complete, i.e., if additional experiment types, methods, or libraries shall be supported, the vocabulary needs to be extended accordingly.

Figure 1 (right-hand side) shows the multiple layers of decision-making required in automatic experiment generation and the involvement of the annotations. First, an experiment type is selected by the user. Here, we currently focus on parameter scanning and simulation-based optimization. Depending on the selected experiment type, method-specific annotations are evaluated to infer which particular method to use. E.g., by providing the annotation `number_of_sample_points` the Latin hypercube design can be distinguished from a full factorial scan since the sample size of the latter is given implicitly by the number of discrete levels in each dimension (this is included in the parameter information). Next, further optional and non-optional annotations are assessed. For a full factorial parameter scan, e.g., at least information about the parameters including lower and upper bounds for the parameter values and a step size for the scan (which determines the number of levels) are needed, whereas in a Latin hypercube design the step size is calculated from the number of points. The other annotations are optional, i.e., default values can be used instead when assembling and filling in the code templates during the next step.

So far, the annotated information is directly related to arguments required by the generated code. Future versions of the notebook extension may support annotations based on external, domain-specific ontologies and consider those in the decision-making process. Prominent examples of this are the various

Table 1: Annotation vocabulary relevant in designing a parameter scan.

Term	Example	Description
@parameter	a=[1.0, 2.0, 0.1] # @parameter b=[1.0, 5.0, 1.0] # @parameter	Specification of a model parameter for a full factorial scan. Requires a parameter name and a triple consisting of a lower bound and upper bound on the parameter values as well as a stepsize used for sampling between these bounds. At least one parameter needs to be annotated.
	a=[1.0, 2.0] # @parameter b=[1.0, 5.0] # @parameter	Parameter specification for a Latin hypercube design. Only lower and upper bound are required.
@output_variable	exampleOutput @output_variable	Name of the desired output variable. Multiple output variables may be annotated.
@output_file	'examplePath.csv' @output_file	Path to a file where the results of the simulation experiment will be recorded.
@exception	'a' * 'b' $\leq$ 50 @exception	Parameter configurations that shall not be considered in the experiment. Specified as a boolean expression about parameters. Arbitrary many exceptions may be declared.
@number_of_ sample_points	100 @number_of_sample_points	Number of samples to be used in a Latin hypercube design.

controlled vocabularies in systems biology (Courtot et al. 2011). Annotations may, e.g., be used for choosing a simulation algorithm and thus for parameterizing the `run_simulation` function. Moreover, an ontology about the parameters and variables of a domain may be exploited to automatically retrieve parameter ranges from online sources. In addition, an ontology that, e.g., formally describes the various ways of conducting a parameter scan and their advantages may assist in automatically choosing the right methodology for the requested simulation experiment (Wilsdorf et al. 2021). Finally, being able to reason about the overall process of a simulation study, including the goals, experiments and analyses conducted, satisfied and unsatisfied requirements, etc., would open up ample possibility for generating simulation experiments depending on what phase of the simulation study we are currently in, be it model building, calibration, validation, or analysis (Wilsdorf et al. 2023).

3.5 Code Generation and Execution

We have developed code templates that can be flexibly combined to accomodate the needs of the selected experiment type and inferred method. A code template consists of a fixed code skeleton and template variables. Based on the information gathered during the previous steps, the necessary code templates are combined and the extracted information is inserted into the variables of the templates. The generated code is automatically entered in a new cell of the Jupyter notebook. There the code can be inspected by the modeler and possibly modifications or extensions can be made. If the modeler is satisfied with the simulation experiment, it can be run via the standard Jupyter Notebook functionality.

Figure 3: Crossing implemented in the microscopic traffic simulation.

3.5.1 Installation and Configuration

To run our software, we assume that Python 3.9+ and Juypter are installed. To install the extension, the software prototype needs to be downloaded from https://github.com/pwilsdorf/nbSimGen and saved in the `nbextensions` folder of the local Jupyter installation. The extension can be enabled by running the commands `jupyter contrib nbextensions install` and `jupyter nbextension enable nbSimGen/main`. Next, the notebook server can be started by executing `juypter notebook` from the command line. On the Extensions tab in the Jupyter frontend, the extension *nbSimGen* should be displayed as enabled. A user can now create a new notebook and select the kernel for that notebook (in our case, Python is supported). Alternatively, one may continue the work on an existing notebook, e.g., the one from our case study.

4 CASE STUDY: MICROSCOPIC TRAFFIC SIMULATION

In this study, a custom microscopic traffic simulator and model shall be built using Python and Jupyter Notebook. The simulator is a hybrid simulator that combines numerical simulation based on the intelligent driver model (Treiber and Kesting 2013) with discrete events scheduling times of arrivals and actions (e.g., turns) of vehicles using an exponential distribution. The model represents an intersection on Rostock University campus (54°04'28.9"N 12°06'13.8"E). A visualization of the crossing is shown in Figure 3. The Jupyter notebook extension *nbSimGen* shall be used to support the conduction of simulation experiments during this simulation study, and thus help the modeler to analyze the effect of various traffic light configurations. The code of the extension and the case study notebook are available on GitHub (https://github.com/pwilsdorf/nbSimGen).

At the beginning of the simulation study, the modeler formulates the research question, as well as assumptions on the parameter values as part of the **conceptual modeling**. If the input parameters are not set specifically, default values for these parameters are used, which correspond to the values provided by the Rostock Transportation Authority. Figure 4 shows a screenshot of the Jupyter notebook with parts of the conceptual model defined in Markdown cells as well as a code cell. As simulation output, the measure `longest_traffic_jam` is selected and marked using the annotation language. The modeler assumes that two of the seven traffic lights have the most impact, and thus the parameters `k1_green_time` and `k3_green_time` are specified and annotated for further investigation. Based on traffic light data and general constraints of the overall traffic network, upper and lower bounds could be assigned to these parameters.

Now that the essential inputs and constraints have been defined, the modeler enters the stage of the actual **model building** and creates several code cells to compose the model, comprising the traffic light

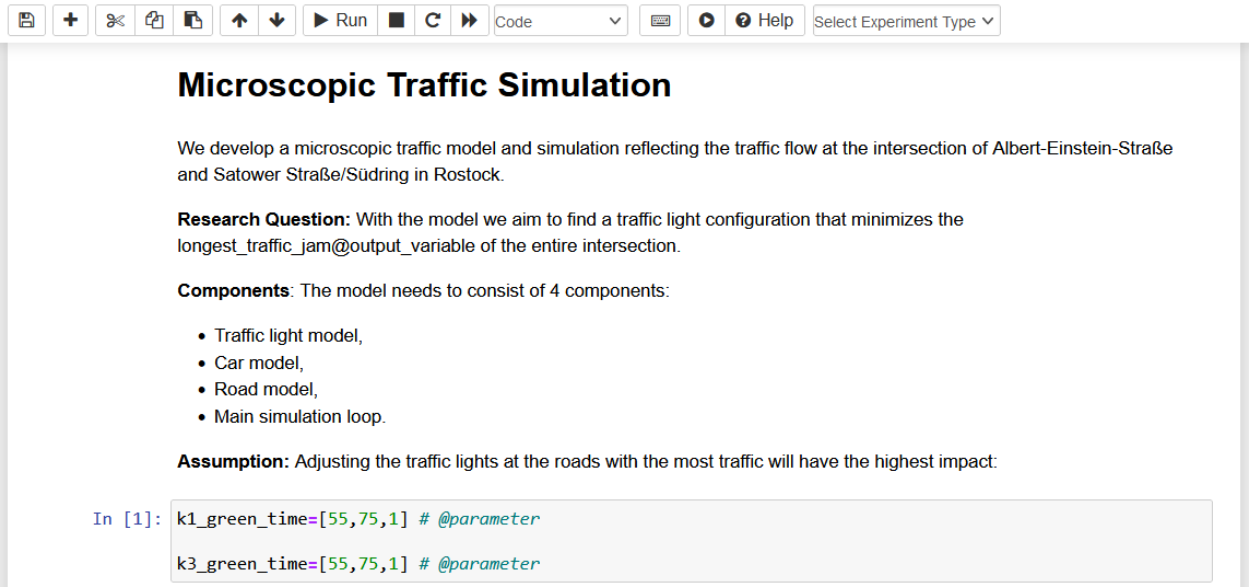

Figure 4: Research question and assumptions annotated with the vocabulary of *nbSimGen*.

model, car model, road model, and main simulation loop. In our small case study, both model and simulator code are implemented as code cells in the Jupyter notebook.

Once the essential features have been implemented into a running model, analyses can be conducted. For this, the output path is specified and annotated using our vocabulary. As a first simulation experiment, the user wants to start with a **parameter scan** to get an overview of the parameter space, and thus selects this experiment type in the user interface of the notebook extension. This triggers the experiment generation of `nbSimGen`, which will now search the notebook for information to be inserted into an parameter scan. Figure 5 shows the generated code of this two-dimensional full-factorial parameter scan and the output produced by running the experiment as well as a visualization of the results as a contour map created by the modeler. Future versions of the notebook extension may also automatically generate a suitable plot for the selected experiments. Here additional annotation vocabulary may be taken into account, e.g., for specifying the units of the inputs and outputs to be used in the labels of the plot.

The shown plot demonstrates the complex response surface of this model that makes it difficult to identify the optimum. Although it provides a first intuition, further analysis is required. In conclusion to those results, the user decides to narrow the parameter bounds before attempting to search more closely for the best traffic light configuration using an **optimization** experiment. Taking the updated parameter bounds into account, a new simulation experiment can be generated based on the SciPy Optimizer library. The code and results of this experiment can be viewed in our GitHub repository.

Based on the results of these two simulation experiments, the modeler now **evaluates and refines the (conceptual) model**. E.g., the modeler wants to add more realism to the model by extending it with different driving styles and capabilities (including different reaction times and driving errors). Thereafter, it would be interesting to **repeat** the earlier experiments to determine the effect of having more variability in the driving behavior. Furthermore, other ouptut measures might be analyzed or optimized, e.g., the overall throughput at the crossing. If all the required information was annotated accordingly during the model extension, then the next simulation experiments can again be generated automatically.

5 CONCLUSIONS

We presented a Jupyter Notebook extension, called *nbSimGen*, that supports the automatic generation and execution of simulation experiments. We thereby provide the first dedicated support for simulation

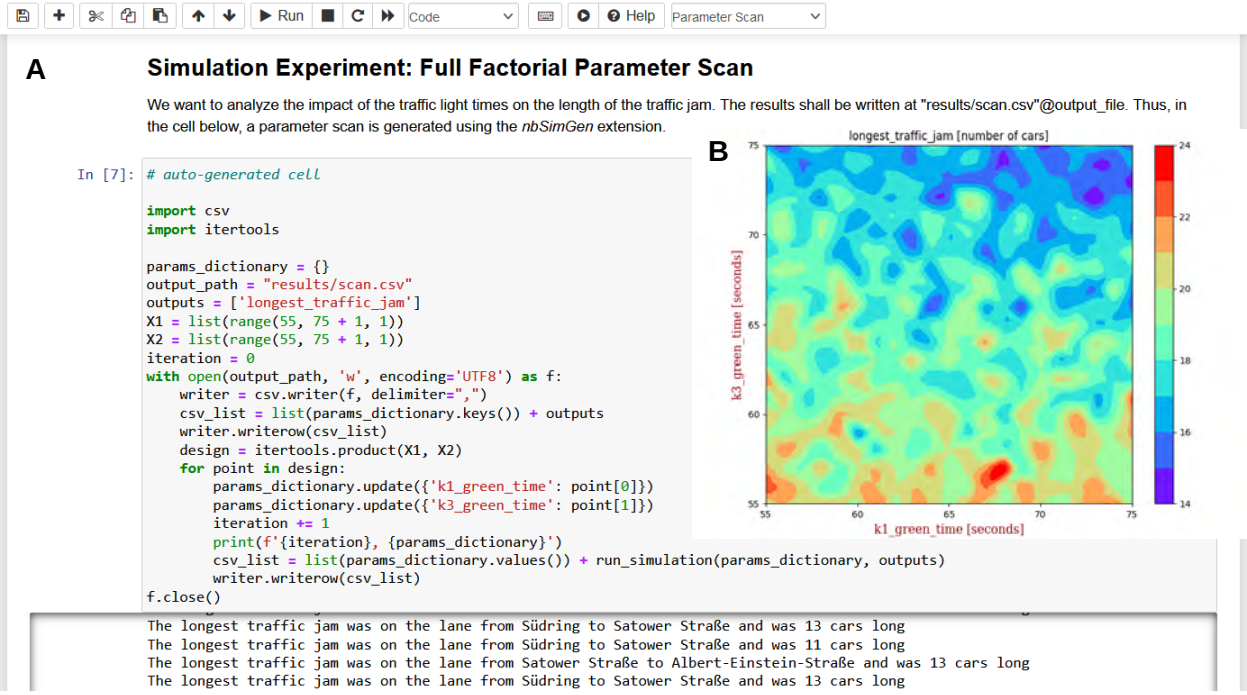

Figure 5: A: Jupyter notebook with generated full factorial parameter scan and simulation output. B: Plot of the simulation results.

experiments in Jupyter Notebook. The notebook extension was applied in a case study of microscopic traffic simulation where first a parameter scan and later an optimization experiment could be generated.

Our approach supports modelers in conducting their simulation studies more efficiently and reduces errors in the specification of simulation experiments. The tool grants access to various types of simulation experiments, different methods and libraries, which we deem especially appealing for novice modelers. Moreover, being implemented in Jupyter, the experiment generation tool profits from the clean, easy-to-use UI design. The code generation approach allows modelers to create tailored solutions by combining and extending automatically generated parts. This also directly improves transparency and reproducibility of the calculations. By supporting simulation experiments via a Jupyter Notebook extension, we also strengthen Jupyter as a modeling and simulation tool and thus modeling and simulation as one discipline of data science.

There are ample possibilities for extending this concept and first prototype. Currently, annotations and rules are kept domain-agnostic to work with a variety of simulation studies. Taking domain-specific knowledge into account would, however, enable support for specialized notebooks, such as Tellurium for systems biology (Medley et al. 2018). Also, the flexible and extensible design of our notebook extension facilitates exporting the developed concept to alternative notebook systems or common IDEs. In addition, we are planning to enrich the modeling notebooks with information required by reporting guidelines such as ODD (Grimm et al. 2020), TRACE (Grimm et al. 2014) or STRESS (Monks et al. 2019) and equip our tool and the annotation language accordingly. For larger projects, linking the notebooks with version control would be practical, i.e., code could be generated either in a new code cell or in a separate file, e.g., in a GitHub repository. So far, we use a controlled vocabulary for annotating specific information in the notebook. This is, however, associated with additional effort required from the users. Investigating the capabilities of natural language processing to identify those parts automatically will therefore be an important next step. The annotation vocabulary we developed may then be used in training the machine

learning models as well as for process mining in the notebooks. The discovered workflows may not only lend support for simulation experiments but also during model building.

ACKNOWLEDGMENTS

This research was funded by the German Research Foundation grant 320435134 “GrEASE”.

REFERENCES

Ayllón, D., S. F. Railsback, C. Gallagher, J. Augusiak, H. Baveco, U. Berger, S. Charles, R. Martin, A. Focks, N. Galic et al. 2021. “Keeping Modelling Notebooks with TRACE: Good for You and Good for Environmental Research and Management Support”. *Environmental Modelling & Software* 136:104932.

Balci, O. 2012. “A Life Cycle for Modeling and Simulation”. *Simulation* 88(7):870–883.

Beg, M., J. Taka, T. Kluyver, A. Konovalov, M. Ragan-Kelley, N. M. Thiéry, and H. Fangohr. 2021. “Using Jupyter for Reproducible Scientific Workflows”. *Computing in Science & Engineering* 23(2):36–46.

Bergmann, F. T., R. Adams, S. Moodie, J. Cooper, M. Glont, M. Golebiewski, M. Hucka, C. Laibe, A. K. Miller, D. P. Nickerson et al. 2014. “COMBINE Archive and OMEX Format: One File to Share all Information to Reproduce a Modeling Project”. *BMC Bioinformatics* 15(1):1–9.

Carvalho, L. A., K. Belhajjame, and C. B. Medeiros. 2016. “Converting Scripts into Reproducible Workflow Research Objects”. In *2016 IEEE 12th International Conference on E-Science (e-Science)*, 71–80. IEEE.

Courtot, M., N. Juty, C. Knüpfer, D. Waltemath, A. Zhukova, A. Dräger, M. Dumontier, A. Finney, M. Golebiewski, J. Hastings, S. Hoops, S. Keating et al. 2011. “Controlled Vocabularies and Semantics in Systems Biology”. *Molecular Systems Biology* 7(1):543.

Cunha, R. L., L. C. V. Real, R. Souza, B. Silva, and M. A. Netto. 2021. “Context-aware Execution Migration Tool for Data Science Jupyter Notebooks on Hybrid Clouds”. In *2021 IEEE 17th International Conference on eScience (eScience)*, 30–39. IEEE.

Drosos, I., T. Barik, P. J. Guo, R. DeLine, and S. Gulwani. 2020. “Wrex: A Unified Programming-by-example Interaction for Synthesizing Readable Code for Data Scientists”. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 1–12.

Fangohr, H., M. Beg, M. Bergemann, V. Bondar, S. Brockhauser, C. Carinan, R. Costa, C. Fortmann, D. F. Marsa, G. Giovanetti et al. 2019. “Data Exploration and Analysis with Jupyter Notebooks”. In *17th Biennial International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS)*, Volume 19, 799–806.

Grimm, V., J. Augusiak, A. Focks, B. M. Frank, F. Gabsi, A. S. Johnston, C. Liu, B. T. Martin, M. Meli, V. Radchuk et al. 2014. “Towards Better Modelling and Decision Support: Documenting Model Development, Testing, and Analysis using TRACE”. *Ecological Modelling* 280:129–139.

Grimm, V., S. Railsback, C. Vincenot, U. Berger, C. Gallagher, D. Deangelis, B. Edmonds, J. Ge, J. Giske, J. Groeneveld, A. Johnston, A. Milles et al. 2020, January. “The ODD Protocol for Describing Agent-based and other Simulation Models: A Second Update to Improve Clarity, Replication, and Structural Realism”. *Journal of Artificial Societies and Social Simulation* 23(2).

Head, A., F. Hohman, T. Barik, S. M. Drucker, and R. DeLine. 2019. “Managing Messes in Computational Notebooks”. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 1–12.

Kluyver, T., B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, and others. 2016. “Jupyter Notebooks-A Publishing Format for Reproducible Computational Workflows”. In *ELPUB*, 87–90.

Lorig, F. 2019. *Hypothesis-Driven Simulation Studies*. Springer.

Matloff, N. 2008. “Introduction to Discrete-Event Simulation and the Simpy Language”. *Davis, CA. Dept of Computer Science. University of California at Davis.* 2(2009):1–33.

Medley, J. K., K. Choi, M. König, L. Smith, S. Gu, J. Hellerstein, S. C. Sealfon, and H. M. Sauro. 2018. “Tellurium Notebooks—An Environment for Reproducible Dynamical Modeling in Systems Biology”. *PLOS Computational Biology* 14(6):e1006220.

Monks, T., C. S. M. Currie, B. S. Onggo, S. Robinson, M. Kunc, and S. J. E. Taylor. 2019. “Strengthening the Reporting of Empirical Simulation Studies: Introducing the STRESS Guidelines”. *Journal of Simulation* 13(1):55–67.

Mooers, B. H. M. 2021. “A PyMOL Snippet Library for Jupyter to Boost Researcher Productivity”. *Computing in Science & Engineering* 23(2):47–53.

Mustafa, T. A., B. König-Ries, and S. Samuel. 2023. “MLProvCodeGen: A Tool for Provenance Data Input and Capture of Customizable Machine Learning Scripts”. In *BTW 2023*, edited by B. König-Ries, S. Scherzinger, W. Lehner, and G. Vossen: Gesellschaft für Informatik e.V.

Naldi, A., C. Hernandez, N. Levy, G. Stoll, P. T. Monteiro, C. Chaouiya, T. Helikar, A. Zinovyev, L. Calzone, S. Cohen-Boulakia et al. 2018. "The CoLoMoTo Interactive Notebook: Accessible and Reproducible Computational Analyses for Qualitative Biological Networks". *Frontiers in Physiology* 9:680.

Pimentel, J. F., L. Murta, V. Braganholo, and J. Freire. 2021. "Understanding and improving the Quality and Reproducibility of Jupyter Notebooks". *Empirical Software Engineering* 26(4):65.

Randles, B. M., I. V. Pasquetto, M. S. Golshan, and C. L. Borgman. 2017. "Using the Jupyter Notebook as a Tool for Open Science: An Empirical Study". In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, 1–2.

Robinson, S. 2017. "A Tutorial on Simulation Conceptual Modeling". In *Proceedings of the 2017 Winter Simulation Conference*, edited by W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, 565–579. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Rule, A., A. Birmingham, C. Zuniga, I. Altintas, S.-C. Huang, R. Knight, N. Moshiri, M. H. Nguyen, S. B. Rosenthal, F. Pérez, and P. W. Rose. 2019, 07. "Ten Simple Rules for Writing and Sharing Computational Analyses in Jupyter Notebooks". *PLOS Computational Biology* 15(7):1–8.

Ruscheinski, A., K. Budde, T. Warnke, P. Wilsdorf, B. C. Hiller, M. Dombrowsky, and A. M. Uhrmacher. 2018. "Generating Simulation Experiments based on Model Documentations and Templates". In *Proceedings of the 2018 Winter Simulation Conference*, edited by M. Rabe, A. A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, 715–726. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

Ruscheinski, A., and A. Uhrmacher. 2017. "Provenance in Modeling and Simulation Studies—Bridging Gaps". In *Proceedings of the 2017 Winter Simulation Conference*, edited by W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, 872–883. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Sargent, R. G. 2013. "Verification and Validation of Simulation Models". *Journal of Simulation* 7(1):12–24.

Savira, P., T. Marrinan, and M. E. Papka. 2021. "Writing, Running, and Analyzing Large-scale Scientific Simulations with Jupyter Notebooks". In *2021 IEEE 11th Symposium on Large Data Analysis and Visualization (LDAV)*, 90–91.

Treiber, M., and A. Kesting. 2013. "Traffic flow dynamics". *Traffic Flow Dynamics: Data, Models and Simulation, Springer-Verlag Berlin Heidelberg*:983–1000.

Wilsdorf, P., N. Fischer, F. Haack, and A. M. Uhrmacher. 2021. "Exploiting Provenance and Ontologies In Supporting Best Practices For Simulation Experiments: A Case Study On Sensitivity Analysis". In *Proceedings of the 2021 Winter Simulation Conference*, edited by S. Kim, B. Feng, K. Smith, S. Masoud, Z. Zheng, and C. S. andM. Loper, 1–12. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

Wilsdorf, P., J. Heller, K. Budde, J. Zimmermann, T. Warnke, C. Haubelt, D. Timmermann, U. van Rienen, and A. M. Uhrmacher. 2022. "A Model-Driven Approach for Conducting Simulation Experiments". *Applied Sciences* 12(16):7977.

Wilsdorf, P., A. Wolpers, J. Hilton, F. Haack, and A. Uhrmacher. 2023. "Automatic Reuse, Adaption, and Execution of Simulation Experiments via Provenance Patterns". *ACM Transactions on Modeling and Computer Simulation* 33(1–2).

Yin, P., W.-D. Li, K. Xiao, A. Rao, Y. Wen, K. Shi, M. Catasta, H. Michalewski, A. Polozov, and C. Sutton. 2022. "Natural Language to Code Generation in Interactive Data Science Notebooks". *arXiv Preprint*.

AUTHOR BIOGRAPHIES

PIA WILSDORF is a Ph.D. candidate and research associate in the Modeling and Simulation group at the University of Rostock. Her email address is pia.wilsdorf@uni-rostock.de.

ANTON WILLY KIRCHHÜBEL is a Bachelor's student of Computer Science at the University of Rostock. His email address is anton.kirchhuebel@gmx.de.

ADELINDE M. UHRMACHER is a Professor at the Institute for Visual and Analytic Computing, University of Rostock, and head of the Modeling and Simulation group. Her email address is adelinde.uhrmacher@uni-rostock.de.